Funktionale Programmierung mit Java

Manuel Mauky & Max Wielsch

Saxonia Systems
So geht Software.

Manuel Mauky
Software Architect

manuel.mauky@saxsys.de
http://lestard.eu
@manuel_mauky

Max Wielsch
Software Engineer

max.wielsch@saxsys.de
http://max-wielsch.blogspot.com
@simawiel

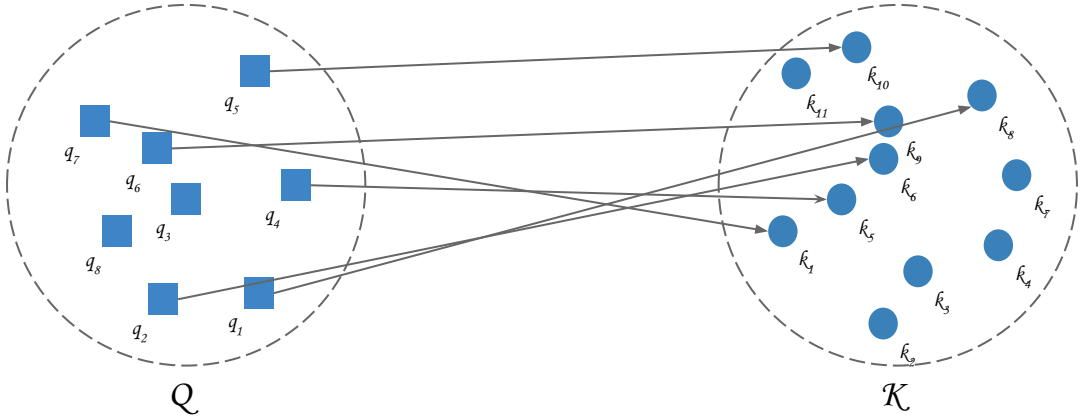
SPEAKER

Was ist “funktionale Programmierung”?

Programmieren mit Funktionen!

Was ist eine Funktion?

Eine Funktion ist eine Beziehung zwischen zwei Mengen, die jedem Element der einen Menge genau ein Element der anderen Menge zuordnet. (wikipedia)



$f(q_1) = k_8$
$f(q_2) = k_9$
$f(q_3) = \text{n. d.}$
$f(q_4) = k_5$
$f(q_5) = k_{10}$
...

Funktion in der Programmierung

Ein Programmkonstrukt, welches

- Argumente entgegen nimmt und
- ein Ergebnis liefert

Ist das eine Funktion?

```
int square(int a) {
    return a * a;
}
```

Ist das eine Funktion?

```
String greet(Person person) {
    return “Hello ” + person.name;
}
```

Ist das eine Funktion?

```
String greet(Person person) {
    return “Hello “ + person.name + “, todays date is “
        + new Date();
}
```

Ist das eine Funktion?

```
String greet(Person person) {
    String message = “Hello “ + person.name;
    System.out.println(message);
    return message;
}
```

Seiteneffekte

Alles, was nicht explizit Argument oder Rückgabewert ist, ist ein “Seiteneffekt” (engl. “side effect”)

- IO: z. B. System.out.println, Datenbankzugriff, Dateien lesen
- Zugriff oder Veränderungen von globalen Variablen

Pure Function

Funktionale Programmierung versucht Seiteneffekte zu vermeiden.

“pure Funktion” = Funktion ohne Seiteneffekte

- Rückgabewert ist ausschließlich von Eingabewerten abhängig
- idempotent: mehrfaches Aufrufen mit selbem Parameter führt immer zum selben Rückgabewert
- Referenzielle Transparenz: Wert eines Ausdrucks hängt nur von der Umgebung ab, nicht vom Zeitpunkt der Auswertung

```
String greet(Person person) {
    return “Hello “ + person.name + “, todays date is “
        + new Date();
}
```

```
// vorher
String greet(Person person) {
   return “Hello “ + person.name + “, todays date is “
      + new Date();
}
// nachher
String greet(Person person, Date date) {
   return “Hello “ + person.name + “, todays date is “
      + date;
}
```

Pure Function?

```
String greet(Person person) {
    String message = “Hello “ + person.name;
    person.setName(“Hugo”);
    return message;
}
```

Immutable Data Structures

```
public class Person {
   private final String name;

   public Person(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
   public Person setName(String newName) {
      return new Person(newName);
   }
}
```

Immutable Data Structures generieren

z. B. http://immutables.github.io/

```
@Value.Immutable
public abstract class Person {
   public abstract String name();
}

Person p = ImmutablePerson.builder()
        .name("Ivan Immutable")
        .build();

System.out.println(greet(p));
```

Immutable Data Structures

Weitere Libraries liefern unveränderliche und auch “lazy” Datenstrukturen für Collections:

- https://github.com/google/guava
- https://github.com/bodar/totallylazy
- https://github.com/functionaljava/functionaljava
- https://github.com/krukow/clj-ds
- ...

Was ist der Nutzen?

Pure Functions

- Nachvollziebarkeit
- sicherere Refactorings
- Komposition

Immutable Data Structures

- automatisch threadsafe
- Unterscheidung zwischen “Werten” und “Instanzen” fällt weg

Was macht die Funktion?

```
String something(Person person) { … }
```

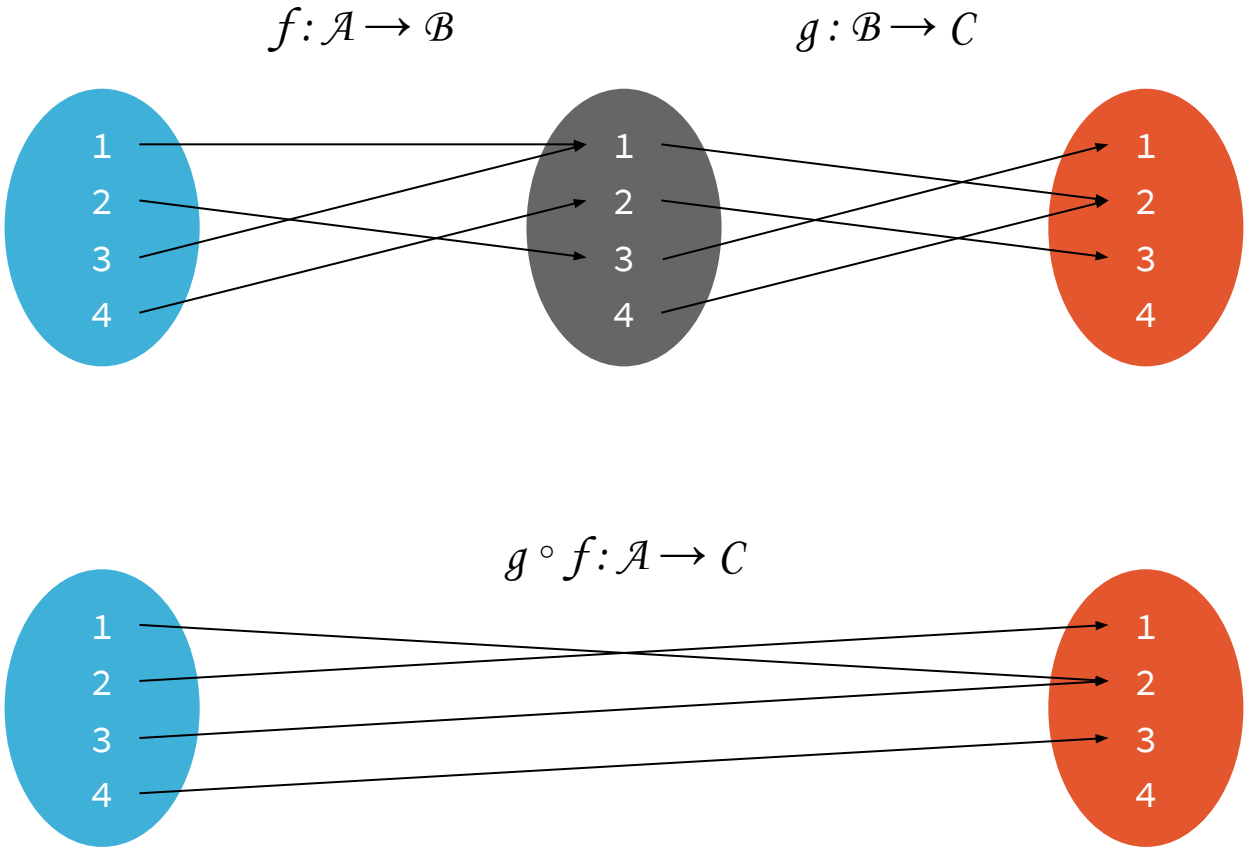
Java?

Ja man kann pure Funktionen in Java schreiben,

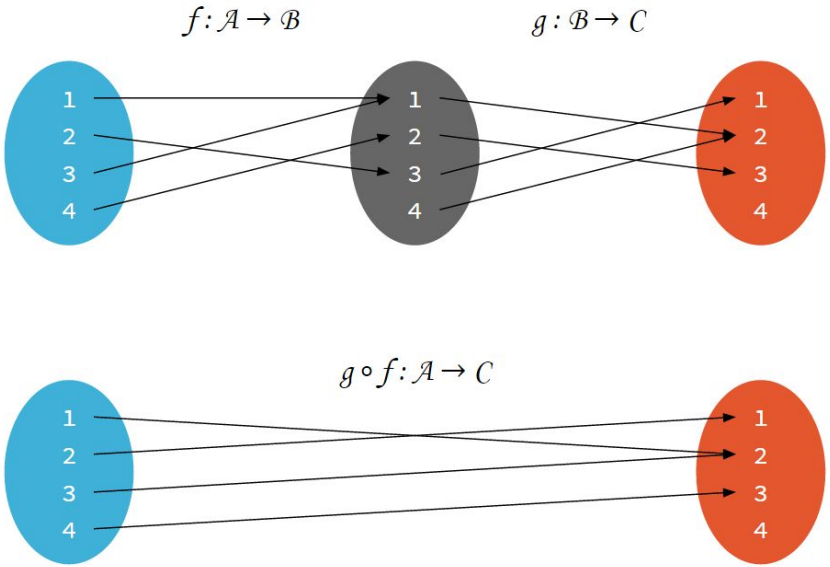
aber:

- es gibt keine Zusicherung für Seiteneffektfreiheit
- es gibt keine Zusicherung für Immutable Data Structures (Java 9?)

FUNKTIONS KOMPOSITION


$f : \mathcal{A} \to \mathcal{B}$
$g : \mathcal{B} \to C$
1
2
3
4
1
2
3
4
1
2
3
4
$g \circ f : \mathcal{A} \to C$
1
2
3
4
1
2
3
4

$$g(f(x)) = (g \circ f)(x)$$


$f: \mathcal{A} \to \mathcal{B}$
$g: \mathcal{B} \to C$
1
2
3
4
1
2
3
4
1
2
3
4
$g \circ f: \mathcal{A} \to C$
1
2
3
4
1
2
3
4

In Java:

```
Function<String,Integer> f = s -> s.length();
Function<Integer,Integer> g = i -> i * i;

g.apply(f.apply(“Hallo”)); // 25

Function<Integer, Integer> h = g.compose(f);
h.apply(“Hallo”); // 25
```

```
kosten :: Produkt -> Euro

euroInDollar :: Euro -> Dollar

kostenInDollar :: Produkt -> Dollar
kostenInDollar = euroInDollar . kosten
```

Einschränkungen in Java?

- Syntax-Overhead
- Function, Consumer, Predicate, etc. sind nicht gut kombinierbar

Unzureichende Java-API

```
Function<Integer, Integer> timesTwo = x -> x * 2;

Consumer<Integer> print = System.out::println;

Supplier<Integer> random = () -> new Random().nextInt(100);

// Wunschvorstellung
Supplier<Integer> randomTimesTwo = timesTwo.compose(random);

SideEffect compose = print.compose(timesTwo.compose(random));
```

Unzureichende Java-API

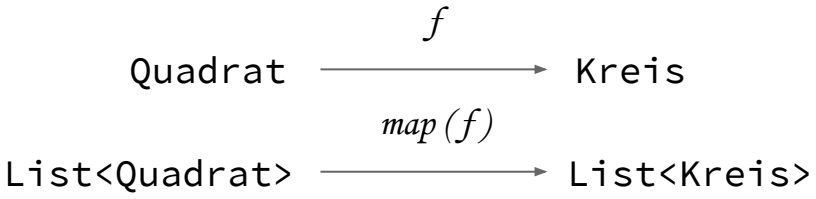
```
Predicate<Integer> isEven = x -> (x & 1) == 0;

Function<Integer, Integer> timesTwo = x -> x * 2;

// Wunschvorstellung
Predicate<Integer> isTimesTwoEven = isEven.compose(timesTwo);  // ;-)
```

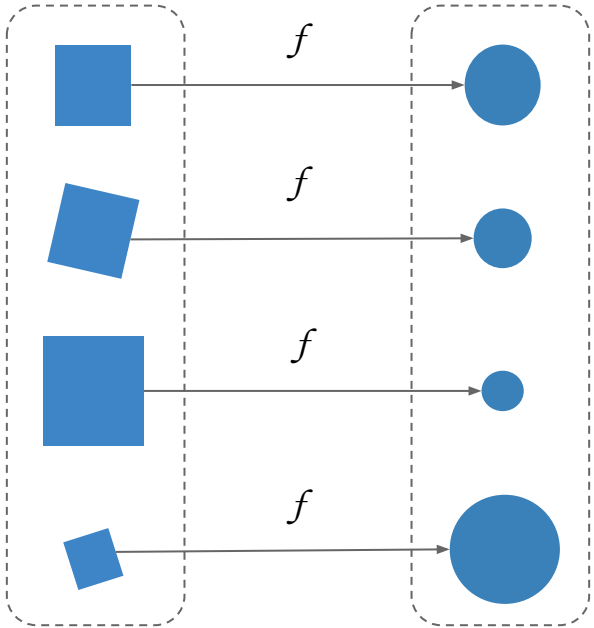
ANWENDUNG VON FUNKTIONEN

Anforderung:

- “Gebäude” besitzt mehrere “Etagen”, “Etage” besitzt mehrere “Räume”
- Raum hat Anzahl an Sitzplätzen
- Manche Etagen sind mit Fahrstuhl erreichbar, andere nicht
- Aufgabe: Finde alle Räume mit mehr als 10 Sitzplätzen auf Etagen, die mit Fahrstuhl erreichbar sind

```
List<Raum> nutzbareRäume = new ArrayList<>();
for (Etage etage : gebäude.getEtagen()) {
    if (etage.isFahrstuhlAvailable()) {
        for (Raum raum : etage.getRäume()) {
            if (raum.getSitzplätze().size() > 10) {
                nutzbareRäume.add(raum);
            }
        }
    }
}
```





Quelle: http://techblog.realestate.com.au/how-we-used-category-theory-to-solve-a-problem-in-java

Beispiel in Java

```
List<Person> persons = …
List<String> emailAddresses =
    persons.stream()
        .map(person -> person.getEmailAddress())
        .collect(Collectors::toList);
```

Beispiel in Java

```
List<Person> persons = …
List<String> emailAddresses =
    persons.stream()
        .filter(person -> person.isActive())
        .map(person -> person.getEmailAddress())
        .collect(Collectors::toList);
```

Vorheriges Beispiel mit Streams

```
List<Raum> nutzbareRäume =
    gebäude.getEtagen().stream()
        .filter(Etage::isFahrstuhlAvailable)
        .flatMap(etage -> etage.getRäume().stream())
        .filter(raum -> raum.size() > 10)
        .collect(Collectors::toList);
```

Map vs. FlatMap?

map: für 1 zu 1 Abbildungen

flatMap: für 1 zu N Abbildungen

```
List<List<Raum>> räume // :-(
     = etagen.stream()
          .map(etage -> etage.getRäume())
          .collect(Collectors:toList);

List<Raum> räume      // :-)
     = etage.stream()
          .flatMap(etage -> etage.getRäume().stream())
          .collect(Collectors::toList);
```

Fold / Reduce

Wie viele Sitzplätze existieren auf Etagen mit Fahrstuhl?

```
int count =
    gebäude.getEtagen().stream()
        .filter(Etage::isFahrstuhlAvailable)
        .flatMap(etage -> etage.getRäume().stream())
        .map(raum -> raum.size())
        ?
```

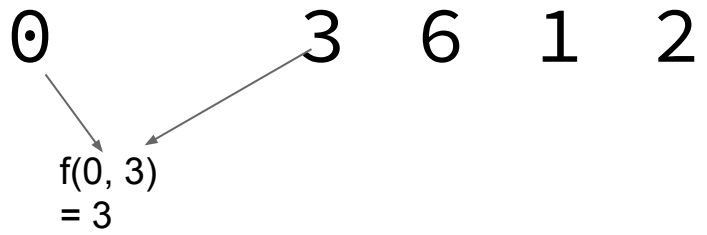
Fold / Reduce

$$f = (a, b) \rightarrow a + b$$

3 6 1 2

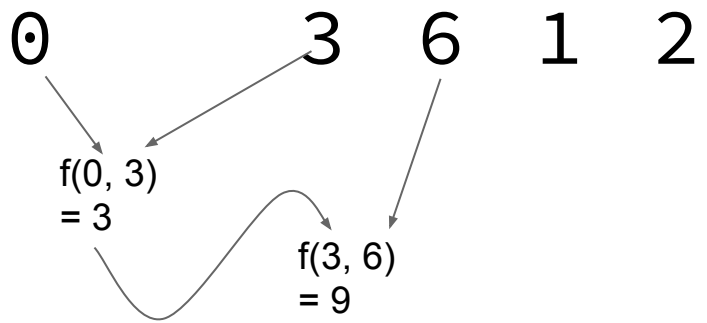
Fold / Reduce

$$f = (a, b) \rightarrow a + b$$



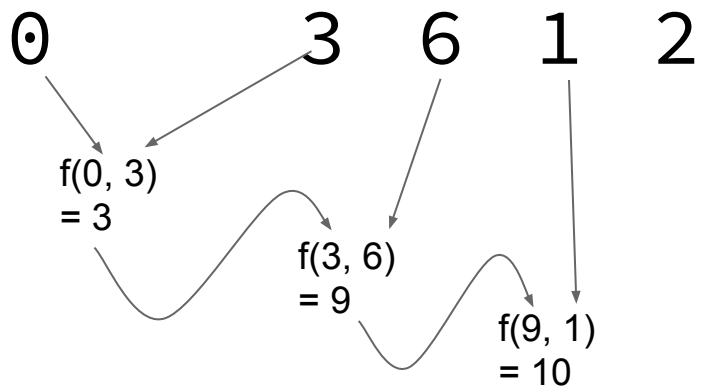
Fold / Reduce

$$f = (a, b) \rightarrow a + b$$



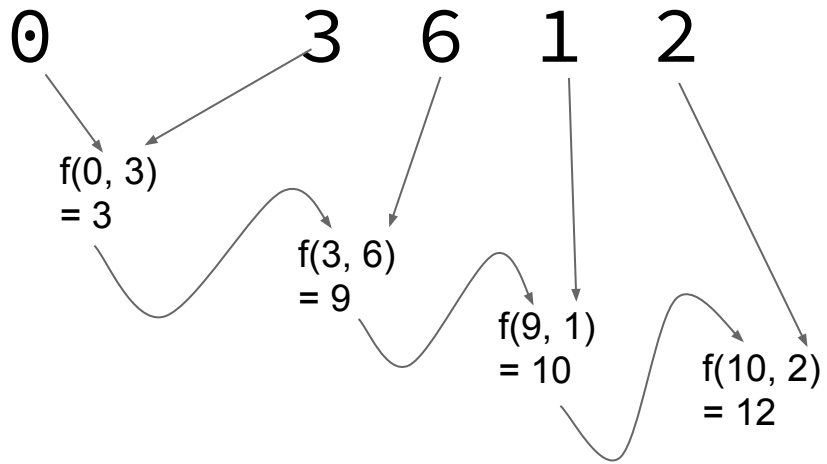
Fold / Reduce

$$f = (a, b) \rightarrow a + b$$



Fold / Reduce

$$f = (a, b) \rightarrow a + b$$



Fold / Reduce

Wie viele Sitzplätze existieren auf Etagen mit Fahrstuhl?

```
int count =
    gebäude.getEtagen().stream()
        .filter(Etage::isFahrstuhlAvailable)
        .flatMap(etage -> etage.getRäume().stream())
        .map(raum -> raum.size())
        .reduce(0, (a,b) -> a + b);
```

Aufgabe:

- Funktion nimmt beliebig viele Funktionen entgegen
- Liefert eine Funktion, die alle Funktionen per function composition kombiniert

```
public <T> Function<T, T> composeAll(Function<T, T> … functions) {
    ...

}
```

Aufgabe:

- Funktion nimmt beliebige viele Funktionen entgegen
- Liefert eine Funktion, die alle Funktionen per function composition kombiniert

```
public <T> Function<T, T> composeAll(Function<T, T> … functions) {
    return Arrays.stream(functions)
        .reduce(Function.identity(), Function::compose);
}
```

Aufgabe: Die Namen aller Räume als kommaseparierter String!

Aufgabe: Die Namen aller Räume als kommaseparierter String!

```
StringJoiner joiner = new StringJoiner(“, “);
gebäude.getEtagen().stream()
    .flatMap(etage -> etage.getRäume().stream())
    .map(raum -> raum.getName())
    .forEach(joiner::add);

String result = joiner.toString();
```

Optional<T>

Optional<T>

```
Optional<Raum> raum =
    gebäude.getEtagen().stream()
        .filter(Etage::isFahrstuhlAvailable)
        .flatMap(etage -> etage.getRäume().stream())
        .filter(raum -> raum.size() > 10)
        .findFirst();
```

Optional<T>

```
class Person {
    String email;

    public Optional<String> getEmail() {
        return Optional.ofNullable(email);
    }
}
```

Optional<T>

```
class Raum {
     String getRaumNummer() {...}
}

Optional<Raum> raum = getRaum();
Optional<String> raumNummer = raum
          .map(r-> r.getRaumNummer());

raumNummer
     .ifPresent(nr -> System.out.println(nr));
```

Optional<T>

```
class Raum {
     Optional<String> getRaumNummer() {...}
}

Optional<Raum> raum = getRaum();
Optional<String> raumNummer = raum
          .flatMap(r-> r.getRaumNummer());

raumNummer
     .ifPresent(nr -> System.out.println(nr));
```

Optional<T>

```
class Raum {
     Optional<String> getRaumNummer() {...}
}

getRaum().flatMap(Raum::getRaumNummer)
         .ifPresent(System.out::println);
```

Reales Beispiel

- “Antrag” kann “Beratungsstelle” besitzen
- “Beratungsstelle” kann “Name1” und “Name2” besitzen (jeweils String)
- Aufgabe:
  - Generiere String für Anzeige, Name1 und Name2 mit Komma getrennt.
  - Falls “Antrag” oder “Beratungsstelle” == null, dann leerer String
  - Komma nur, wenn sowohl “Name1” als auch “Name2” vorhanden sind

```
String result = “”;
if (application != null) {
    ConsultingCenter cc = application.getCC();
    if (cc != null) {
        result += cc.getName1();
        if (cc.getName2() != null) {
            result += “, “ + cc.getName2();
        }
    }
}
```

```
String result = “”;
if (application != null) {
    ConsultingCenter cc = application.getCC();
    if (cc != null) {
        if (cc.getName1() != null && cc.getName2() != null) {
            result = cc.getName1() + “, “ + cc.getName2();
        } else if (cc.getName1() != null) {
            result = cc.getName1();
        } else if (cc.getName2() != null) {
            result = cc.getName2();
        }
    }
}
```

```
StringJoiner joiner = new StringJoiner(", ");
Optional<ConsultingCenter> cc =
    Optional.ofNullable(application).map(Application::getCC);

cc.map(ConsultingCenter::getName1).ifPresent(joiner::add);
cc.map(ConsultingCenter::getName2).ifPresent(joiner::add);

String result = joiner.toString();
```

Gemeinsamkeit Optional und Stream? Monaden!

Monade: Ein “Wrapper/Container/Context” der bestimmten Regeln folgt und funktionale Komposition erlaubt. (stark vereinfacht ;-))

Gemeinsamkeit Optional und Stream? Monaden!

Ein “Wrapper/Container/Context” der bestimmten Regeln folgt und funktionale Komposition erlaubt.

Begriffe:

`unit / return`	`t -> M t`	`Optional.of`	`Stream.of`
`bind`	`M t -> (t -> M u) -> M u`	`Optional.flatMap`	`Stream.flatMap`

Regeln

Regel 1: Left Identity

“Die Anwendung von bind mit einer Funktion f auf eine Monade liefert das gleiche Ergebnis wie die direkte Anwendung von f auf den Basis-Wert der Monade”

```
Function<Integer, Optional<Integer>> timesTwo = x -> Optional.of(x * x);


Optional.of(12).flatMap(timesTwo) == timesTwo.apply(12)
```

Regeln

Regel 2: Right Identity

“Haben wir eine Monade und benutzen bind auf die unit Funktion der Monade erhalten wir die ursprüngliche Monade”

```
Optional.of(12).flatMap(Optional::of) == Optional.of(12)
```

Regeln

Regel 3: Assoziativität

“Werden mehrere Funktionen auf eine Monade mittels bind angewandt, spielt die Reihenfolge keine Rolle”

```
Optional<Integer> monad = Optional.of(12);

Function<Integer, Optional<Integer>> f = n -> Optional.of(n * 2);

Function<Integer, Optional<Integer>> g = n -> Optional.of(n + 5);

Function<Integer, Optional<Integer>> g_f = n -> f.apply(n).flatMap(g);

monad.flatMap(f).flatMap(g) == monad.flatMap(g_f);
```

Monaden in Java

- 3 in JDK: Optional, Stream und CompletableFuture
- Allerdings:
  - kein gemeinsames Interface, daher kein abstrakter Code für alle Monadentypen möglich

```
// in Java nicht möglich:

public <T extends Monad> Integer something(T<Integer> monad) {...}
```

FAZIT

Fazit

- Beschäftigung mit funktionaler Programmierung ist spannend und lehrreich
- besserer Code auch in Java
- Beschäftigung mit “richtiger” funktionaler Sprache trotzdem lohnenswert

Funktional in Java

- Ja, man kann funktionale Konzepte in Java umsetzen und besser verständlichen und testbaren Code schreiben.
- Java 8 bringt Lambdas und Klassen für Funktionale Programmierung. Allerdings könnte die Zusammenarbeit mit verschiedenen Typen besser funktionieren.
- Es sind nicht alle Konzepte in Java technisch unterstützt, teilweise können diese durch Bibliotheken unterstützt werden.

Funktional in Java - Einschränkungen

- Es gibt verschiedene funktionale Typen in Java (monadische Typen, Functional Interfaces, ...), dennoch ist in der Verwendung ein Bruch zwischen diesen zu sehen.
- Trotz Lambdas ist in der Umsetzung immer noch mehr Boilerplate, als in einer reinen funktionalen Sprache.
- Man kann zwar funktional programmieren, aber man wird nicht dazu gezwungen. Die Sprache lässt Seiteneffekte zu, ohne, dass man es dem Code äußerlich ansieht.
  Bei reinen funktionalen Sprachen “steckt innen drin, was außen drauf steht.”
- Beschränktes Typsystem: something(List<T> a) -> cooler wäre: something(T<Integer> a)

Ausblick: Beim nächsten Mal

Haskell

- rein funktional
- Lazy / non-strict Evaluation
- mächtiges Typ-System
- Seiteneffekte explizit über monadische Typen
- Durch “Frege” auch auf der Java-VM nutzbar